Programming Assignment 5
FrequentFlyerAccount

Time due: 9:00 PM, Thursday, March 2nd

Introduction

According to Wikipedia, a frequent flyer account is an airline loyalty program designed to encourage airline customers enrolled in the program to accumulate points which may then be redeemed for air travel or other rewards. Points earned may be based on the class of fare, distance flown on that airline or its partners, or the amount paid. Sometimes, there are other ways to earn points such as by using a co-branded credit card rather than by air travel. Points can be redeemed for air travel, other goods or services, or for increased benefits, such as travel class upgrades, airport lounge access, fast track access, or priority bookings.

Your task

Your assignment is to produce a two classes that work together to simulate PlaneFlights and a FrequentFlyerAccount. In an effort to help you, the design of these two classes will be discussed here. In addition, some sample code has been provided to assist you with this task. Various UML diagrams are shown below to communicate the code you need to create. Please follow the steps outlined below and don't jump ahead until you have finished the earlier step.

First, you will create the class PlaneFlight. This class represents plane trip. Each PlaneFlight object has a passenger name's, a cost, a from and to city as well as the mileage associated with this trip. All five of these parameters are provided to the PlaneFlight constructor. In addition to a constructor, each data member has a public accessor and mutator operation. The constructor and the mutator operations should enforce the following data validation rules:
- a valid cost must be a value of 0 or more. (A zero-cost flight will indicate a free flight earned by a frequent flyer. More on that shortly...). A PlaneFlight should store the value -1 to indicate when an invalid flight cost was attempted to be stored.
- empty string values are not valid for either a FromCity or a ToCity value. Additionally, the specified FromCity and ToCity must be different. A PlaneFlight should ignore and not accept an invalid FromCity or ToCity value. Similarly, the empty string is not valid for a passenger's Name. A PlaneFlight should ignore and not accept an invalid Name value.
- a valid mileage must be a value greater than 0. A PlaneFlight should store the value -1 to indicate when an invalid mileage amount was attempted to be stored.

Please review the class diagram shown here:

PlaneFlight
- mCost : double - mFromCity : string - mToCity : string - mName : string - mMileage : double
+ PlaneFlight(passengerName : string, fromCity : string, toCity : string, cost : double, mileage : double) + getCost( ) : double + setCost(cost : double) : void + getMileage( ) : double + setMileage(mileage : double) : void + getName( ) : string + setName(name : string) : void + getFromCity( ) : string + setFromCity(from : string) : void + getToCity( ) : string + setToCity(to : string) : void

Next, create the FrequentFlyerAccount class. Each FrequentFlyerAccount object has a name associated with the account and its mileage balance. Solely the name parameter is provided to the FrequentFlyerAccount constructor. In the beginning of time, the balance should start at zero. In addition to a constructor, each data member has a public accessor operation. The mileage balance gets increased by adding flights to the account via calls to .addFlightToAccount(...) when the passenger's name matches the frequent flyer account name. .addFlightToAccount(...) should return true when the names match and return false otherwise. Free flights can be redeemed from a FrequentFlyerAccount via calls to .freeFlight(...) which should use the passed parameters to create the desired PlaneFlight with a zero cost, adjusting the mileage balance accordingly. .freeFlight(...) should return true when enough of a mileage balance existed to create a free flight and return false otherwise. A FrequentFlyerAccount can also be used to determine if enough of a mileage balance is available for a desired flight via calls to .canEarnFreeFlight( ) which returns the appropriate boolean answer.

Please review the class diagram shown here:

FrequentFlyerAccount
- mName : string - mBalance : double
+ FrequentFlyerAccount(name : string) + getBalance( ) : double + getName( ) : string + addFlightToAccount(flight : PlaneFlight) : bool + canEarnFreeFlight(mileage : double) bool + freeFlight(from : string, to : string, mileage : double, flight : PlaneFlight &) : bool

For this project, you will create both a .h and .cpp for this class. Write some sample driver code in your main( ) and create assertions to verify that your accessor methods are all working properly. Some sample code is shown below to further document how this class should work.

You are free to create additional public and private methods and data members as you see fit. However, the test cases will only be driving the public methods of the two classes diagrammed here.

The source files you turn in will be these classes and a main routine. You can have the main routine do whatever you want, because we will rename it to something harmless, never call it, and append our own main routine to your file. Our main routine will thoroughly test your functions. You'll probably want your main routine to do the same. If you wish, you may write additional class operation in addition to those required here. We will not directly call any such additional operations directly.

The program you turn in must build successfully, and during execution, no method may read anything from cin. If you want to print things out for debugging purposes, write to cerr instead of cout. When we test your program, we will cause everything written to cerr to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

Please read the posted FAQ for further assistance.

Programming Guidelines

Your program must *not* use any function templates from the algorithms portion of the Standard C++ library or use STL <list> or <vector>. If you don't know what the previous sentence is talking about, you have nothing to worry about. Additionally, your code

must *not* use any global variables which are variables declared outside the scope of your individual functions.

Your program must build successfully under both Visual C++ and either clang++ or g++.

The correctness of your program must not depend on undefined program behavior.

What you will turn in for this assignment is a zip file containing the following 6 files and nothing more:

1. The text files named **PlaneFlight.h** and **PlaneFlight.cpp** that implement the PlaneFlight class diagrammed above, the text files named **FrequentFlyerAccount.h** and **FrequentFlyerAccount.cpp** that implement the FrequentFlyerAccount class diagrammed above, and the text file named **main.cpp** which will hold your main program. Your source code should have helpful comments that explain any non-obvious code.
2. A file named **report.doc** or **report.docx** (in Microsoft Word format), or **report.txt** (an ordinary text file) that contains in addition **your name** and **your UCLA Id Number**:
   - A brief description of notable obstacles you overcame
   - A list of the test data that could be used to thoroughly test your functions, along with the reason for each test. You must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.) Notice that most of this portion of your report can be written just after you read the requirements in this specification, before you even start designing your program.

As with Project 3 and 4, a nice way to test your functions is to use the assert facility from the standard library. As an example, here's a very incomplete set of tests for Project 5. Again, please build your solution incrementally. So I wouldn't run all these tests from the start because many of them will fail until you have all your code working. But I hope this gives you some ideas....

```
#include <iostream>
#include <string>
#include <cassert>

#include "PlaneFlight.h"
#include "FrequentFlyerAccount.h"

using namespace std;

int main()
{
   // sample test code
   PlaneFlight shortleg( "Howard", "LAX", "LAS", 49.00, 285 );

   PlaneFlight longleg( "Howard", "LAS", "NYC", 399.00, 2800 );
```

```
            PlaneFlight sample( "Sample", "Sample", "Sample", 0, 1 );

            FrequentFlyerAccount account( "Howard" );

            assert( shortleg.getFromCity( ) == "LAX" );

            assert( shortleg.getToCity( ) == "LAS" );

            assert( shortleg.getName( ) == "Howard" );

            assert( std::to_string( shortleg.getCost( ) ) == "49.000000" );

            assert( std::to_string( shortleg.getMileage( ) ) == "285.000000" );



            // account balance starts at zero...

            assert( std::to_string( account.getBalance( ) ) == "0.000000" );

            assert( account.getName( ) == "Howard" );

            assert( account.canEarnFreeFlight( 3300.00 ) == false );



            // flights add to an account balance

            assert( account.addFlightToAccount( shortleg ) == true );  // returns true
because the names match

            assert( account.addFlightToAccount( longleg ) == true );   // returns true
because the names match

            assert( std::to_string( account.getBalance( ) ) == "3085.000000" );

            // free flights reduce an account balance

            if (account.canEarnFreeFlight( 285 ))

            {

                assert( account.freeFlight( "LAS", "LAX", 285, sample ) == true );

                // Howard earned a free flight...

                assert( sample.getName( ) == "Howard" );

                assert( std::to_string( sample.getCost( ) ) == "0.000000" );
```

```
            assert( sample.getFromCity( ) == "LAS" );

            assert( sample.getToCity( ) == "LAX" );

            assert( std::to_string( sample.getMileage( ) ) == "285.000000" );

            // account has been reduced for this free flight...

            assert( std::to_string( account.getBalance( ) ) == "2800.000000" );

        }

        else

        {

            assert( false );  // there are enough miles in the account...

        }

        // non-matching names are ignored

        PlaneFlight muffin( "Muffin", "LAX", "Doggie Heaven", 500, 500 );

        assert( account.addFlightToAccount( muffin ) == false );

        assert( std::to_string( account.getBalance( ) ) == "2800.000000" );

        cout << "all tests passed!" << endl;

        return( 0 );

    }
```

G31 Build Commands

```
g31 -c PlaneFlight.cpp
g31 -c FrequentFlyerAccount.cpp
g31 -c main.cpp
g31 -o
runnable    main.o    PlaneFlight.o    FrequentFlyerAccount.o
./runnable
```